Implementation of the Metal Privileged Architecture

by

Fatemeh Hassani

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The privileged architecture of modern computer architectures is expanded through new architectural features that are implemented in hardware or through instruction set extensions. These extensions are tied to particular architecture and operating system developers are not able to customize the privileged mechanisms. As a result, they have to work around fixed abstractions provided by processor vendors to implement desired functionalities. Programmable approaches such as PALcode also remain heavily tied to the hardware and modifying the privileged architecture has to be done by the processor manufacturer. To accelerate operating system development and enable rapid prototyping of new operating system designs and features, we need to rethink the privileged architecture design.

We present a new abstraction called Metal that enables extensions to the architecture by the operating system. It provides system developers with a general-purpose and easy-to-use interface to build a variety of facilities that range from performance measurements to novel privilege models. We implement a simplified version of the Alpha architecture which we call μAlpha and build a prototype of Metal on this architecture. μAlpha is a five-stage pipelined processor with a multi-level cache hierarchy. Lastly, we implement a few facilities in Metal including system calls and transactional memory to show the practicality of Metal.

Acknowledgements

I would like to thank my supervisor, Dr. Ali Mashtizadeh, for his patient guidance and support throughout my degree. I greatly appreciate the time and effort he put into helping me with this research project.

I would also like to thank the thesis committee members, Dr. Bernard Wong and Dr. Samer Al-Kiswany for their insightful comments and feedback.

I owe much gratitude to my friend, Aimilios Tslapatis, for his valuable ideas and help with this project.

I wish to thank my parents for their support and encouragement throughout my study. Most importantly, I want to express my deepest gratitude to my loving and supportive husband, Shervin. Without him, I would not have been able to finish this degree. I dedicate this thesis to him.

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The privileged architecture of a processor provides firmware and operating systems with mechanisms to protect shared resources such as memory, I/O devices, and CPU to enable resource management, partitioning, and security. It consists of instructions, privilege levels, and policies that protect different components of the application stack and define the interface between them. The privileged architecture provides functionalities and hardware facilities required for running the operating system and connecting external devices. For most commercial and research systems the privileged architecture design dates back to the PDP-11 [33] and the rise in popularity of the UNIX operating system. UNIX requires two privilege levels, user level and kernel level. The transitions between these two levels are constrained with system calls.

The privileged architecture is created and expanded using hardware extensions. In this approach, instruction sets need to be extended in each generation of processors to provide primitives for a variety of features such as virtualization, security enclaves, system call interface. Often these primitives are not suitable for all applications and multiple versions of them are added to fit different kinds of applications. Furthermore, these primitives are part of the hardware and cannot be removed once published. As a result, privileged architectures have an inflexible design, and extending them is a complex and time-consuming process.

Operating system researchers and developers need a way to extend the privileged architecture easily to support new operating system designs and features. Furthermore, it is essential to be able to iterate quickly through different approaches to privileged primitives in order to accelerate the evolution of operating systems. As a result, a mechanism for quick expansion of the privileged architecture is desired.

In this thesis, we present Metal, a programmable architecture for privileged architecture extensions. Privileged primitives that are implemented in hardware, closely resemble the software abstraction from which they originate. Metal enables rapid expansion of the privileged architecture through software rather than hardware. Using this simple and powerful tool, operating system developers can build new abstractions and organizations. Metal provides basic blocks that can be used to construct all kinds of high-performance low-level mechanisms related to the privileged architecture. Applications include but are not limited to security models, secure enclaves, and virtualization.

We implement a simplified version of Alpha [34] architecture using Verilog. We implement the Metal programmable privileged architecture on this processor to demonstrate its capabilities and several different applications.

The thesis is organized as follows. In Chapter 2, we describe the Alpha architecture and our implementation of this architecture. We present Metal, the programmable privileged architecture in Chapter 3, and we explain its design and implementation on our alpha processor. In Chapter 4, we evaluate different applications of Metal. Finally, we review the related work in Chapter 5 and our conclusions in Chapter 6.

Chapter 2

Alpha Architecture

This chapter gives an overview of the Alpha architecture and the design of the μAlpha, our implementation of the architecture and its features.

2.1 Overview

Alpha is a 64-bit RISC architecture with 32-bit instructions. Memory operations are only loads and stores and all calculations are performed with register-to-register instructions. Alpha supports 8-bit (byte), 16-bit (word), 32-bit (longword), and 64-bit (quadword) integers. It has 32 integer registers, R0 to R31. All the registers are 64-bit, and R31 is always set to zero. Alpha uses a 64-bit virtual address space and supports both little-endian and big-endian byte addressing.

We use the Alpha architecture because of its simplicity. There are no condition codes, no branch delay slots, no load delay slots, no precise arithmetic exceptions, and no single byte writes to memory in the Alpha architecture [34]. It has a very relaxed memory model that allows for a lot of flexibility in the implementation while requiring compilers to enforce ordering through barriers.

2.2 Instructions

We show the subset of Alpha instructions that μAlpha implements in Appendix A. Floating-point instructions are not implemented in the μAlpha (See [35] for a full list of Alpha instructions).

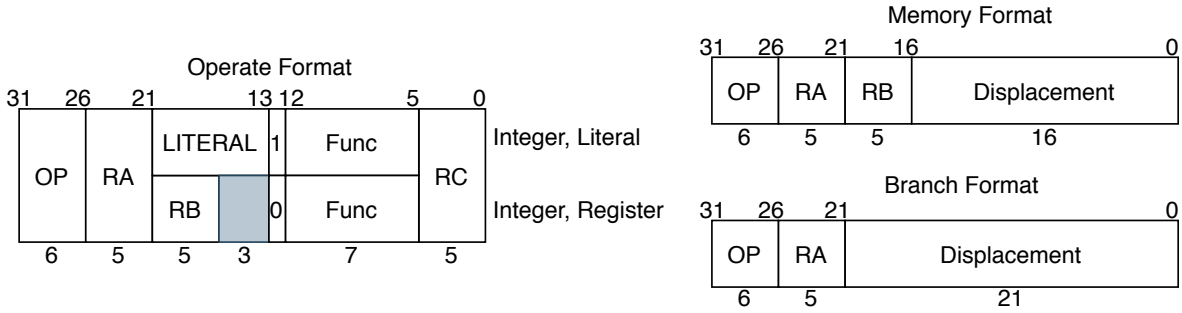

Figure 2.1: Instruction formats in Alpha architecture [34].

All Alpha instructions are 32-bit long and are placed at longword aligned addresses in memory. Appendix A shows the Alpha instructions that our system supports. These instructions can be divided into five categories: integer arithmetic, logical, byte-manipulation, load/store, and branch instructions. We use three instruction formats as shown in Figure 2.1. All of them include a 6-bit opcode field.

The operate format is used for integer arithmetic, logical, and byte manipulation instructions. These instructions contain three 5-bit register number fields (RA, RB, and RC), 7-bit function field, and 8-bit literal field. These instructions calculate `RC = RA operate RB`. The operation is specified by the opcode and the function bits. RB can be replaced with a zero-extended literal if the 13th bit shown in Figure 2.1 is set [35].

The memory format contains two register numbers (RA and RB) and a 16-bit displacement field. It is used for loads, stores, jumps, and some Metal instructions. In load and store instructions, the RA value and the sign-extended displacement are used to calculate the virtual byte address and RB is the source register in stores and destination register in loads. For jump instructions, the RB value and the sign-extended displacement specify the target address, and RA is used to save the program counter (PC) [35].

Branch format includes a 5-bit register number field (RA) and a 21-bit displacement field. This format is used for branch instructions and also two Metal instructions. The displacement is used to calculate the PC-relative target address. In conditional branches, RA is tested against the condition specified by the opcode and in unconditional branches, RA is used in a similar fashion to jump instructions [35].

Metal instructions are explained in section 3.3.3. There is another instruction format in Alpha architecture that is used for PALcode. We do not need this format because Metal replaces PALcode.

2.3 μAlpha Design

Figure 2.2 shows an overview of the μAlpha design with its major functional units. The general design is similar to the Alpha processor presented in [12]. The execution box (*EBox*) is connected to all other components and contains the fetch, decode, and control units. The integer register file (*IRF*) contains the integer register file and data forwarding paths and provides two operands to the IBox. The integer box (*IBox*) is the integer arithmetic and logic unit and produces a 64-bit result that is either a calculated address for accessing the memory or a value that needs to be saved in one of the registers. The memory box (*MBox*) contains the memory reference unit. It gets input from the IRF and the IBox and issues loads and stores to the data cache. Additionally, the MBox delivers the write-back data to the IRF. The *ICache* and *DCache* are instruction and data caches. They both connect to the L2 cache through the bus interface (*BI*). The *MetalMem* is a memory that keeps Metal instructions, which is similar to a microcode ROM in other architectures. Metal design is explained in Chapter 3. The following sections give a detailed description of each functional unit.

2.4 Pipeline Organization

We use a 5-stage pipeline for executing Alpha instructions efficiently. In Figure 2.3 the pipeline is demonstrated with the main functional units. The five stages of the pipeline are described in the following paragraphs.

- **Stage 1: Instruction Fetch**
  In this stage, an instruction is read from either the ICache or the Metal memory based on the range of the PC. Also, the calculation to determine the next PC is performed during this stage. The fetched instruction is passed to the decoder and pipeline controller. The instruction fetch unit is implemented in the Ebox.

- **Stage 2: Register Read**
  The second stage of the pipeline is handled by the IRF. During this stage, the operands are read from the integer register file. The data from the registers, displacement, literal, or the bypass data is saved and passed to the next stage. If the instruction is branch, jump, or Metal enter or exit, the target address is determined during this stage. For conditional branches, this means the register value is also compared against the specified condition.

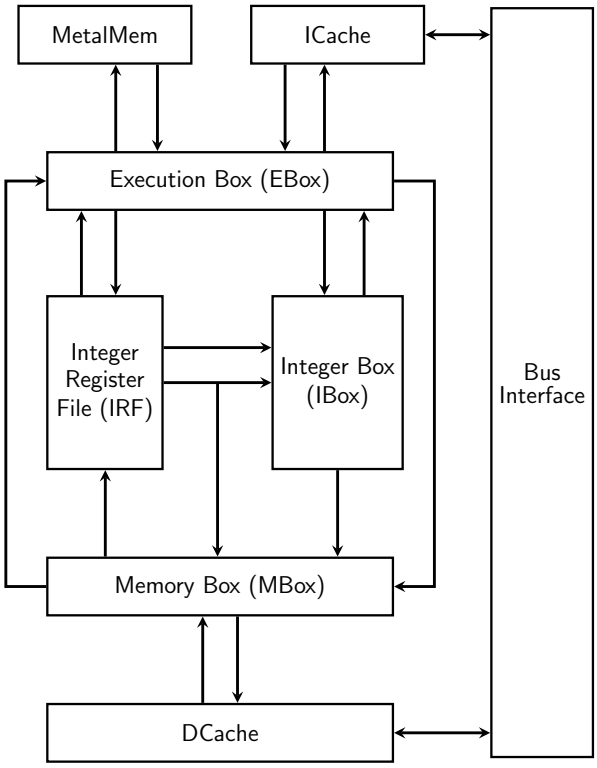

Figure 2.2: μAlpha general design diagram.

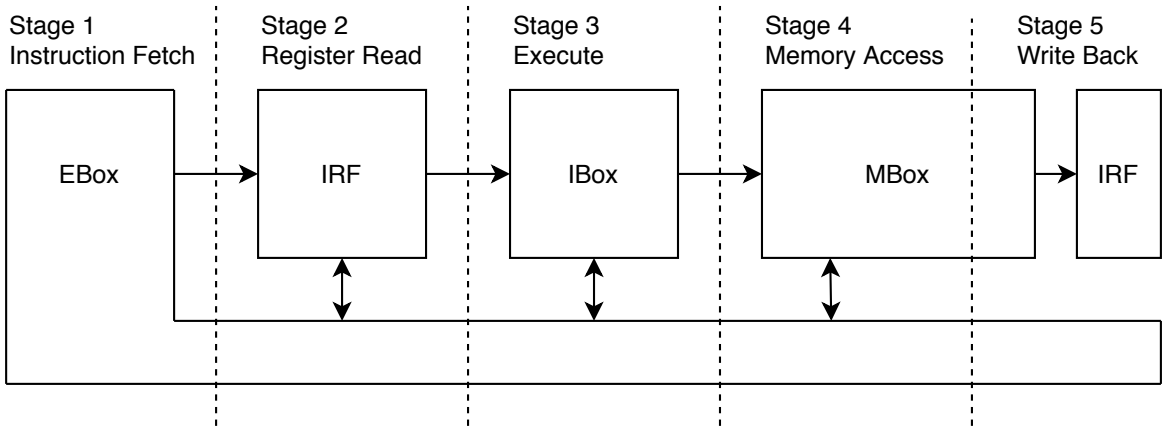

Figure 2.3: Pipeline organization.

- **Stage 3: Execute**
  The execution stage occurs in the IBox where all calculations are performed on the input operands and the result is saved for the next stage.

- **Stage 4: DCache Access**
  This stage is implemented as a part of the MBox. Load and store instructions access the data cache and Metal instructions can read and write the Metal register file during this stage.

- **Stage 5: Write Back**
  In the last stage of the pipeline, instructions write their result into the destination register. The register file is accessed at the same time in stage 2 and 5 of the pipeline. To avoid a resource hazard, the register file performs the write operations in the first half of the cycle for stage 5 and read operations in the second half cycle for stage 2.

2.4.1 Pipeline Hazards

Data Hazards

All instructions in the pipeline are executed in order and the only type of data dependency that can cause a hazard in the pipeline is read after write. The register read is performed in the second half cycle of stage 2 and register write is completed in the first half cycle of stage 5. So, the problem occurs when the destination register of an instruction is the same

as one of the source registers of the immediate next instruction or the one after that. We use data forwarding to ensure the correctness of the execution in these situations.

Control Hazards

We use bubbling to eliminate control hazards that occur after branch, jump, and Metal enter and exit instructions. The pipeline controller inserts a bubble after these instructions, to make time for the target address to be calculated and loaded to the PC.

2.5 EBox: Fetch and Decode

Figure 2.4 shows the internal design of the EBox which can be divided into three sections: instruction fetch logic, instruction decode logic, and pipeline controller.

2.5.1 Instruction Fetch

Instruction fetch logic is shown in Figure 2.5. It contains a Program Counter (PC) table that keeps the virtual addresses of all in-flight instructions in the pipeline. It also contains the logic for updating the PC. Only one instruction per cycle is fetched. If the virtual instruction address is within the last 64KB of the address space, the instruction is fetched from the Metal memory, otherwise, it is fetched from the ICache (*Check Range* unit in Figure 2.4). We explain Metal in more detail in Chapter 3. The next instruction address is calculated based on the category of the previous instruction that is now in Stage 2 of the pipeline. The following list explains the next address calculation after each type of instruction (illustrated as *Mux 1* inputs in Figure 2.5).

- Branch: The comparator tests the value of register RA against the condition specified by the opcode. The PC gets the target address only if the condition is satisfied. To calculate the target address, the displacement bits in the instruction are shifted 2 bits to the left, sign-extended to 64 bits, and then added to the updated PC.
- Jump or Exit Metal: PC gets the target address directly from register RB. The two least significant bits of RB are ignored.
- Enter Metal: The first 6 bits of displacement are shifted 2 bits to the left, sign-extended to 64 bits, and then added to the starting address of Metal memory.
- Other: The next address is simply calculated by adding 4 to the current PC.

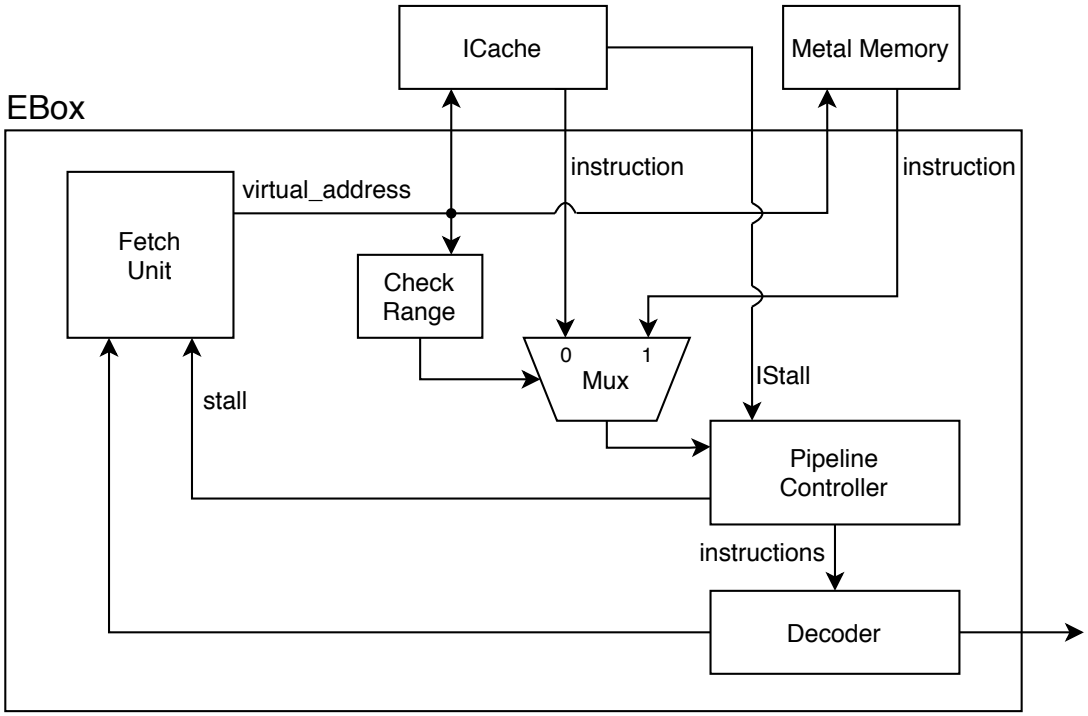

Figure 2.4: EBox block diagram.

2.5.2 Instruction Decode

The decoder consists of a combinational logic that prepares the inputs for other hardware units based on the instructions in the pipeline. Some instruction bits are used directly and others go through a set of logic gates.

2.5.3 Pipeline Controller

The pipeline controller keeps all the in-flight instructions. It contains the hazard detection, the stall logic, and the exception logic. Figure 2.6 shows the pipeline controller diagram. It inserts a NOP instruction after branch, jump, MENTER, and MEXIT instructions. It also inserts a bubble when a load or MRPCR dependency occurs. This type of data hazard is explained in Section 2.6.1. When an exception is generated in stage 3 or 4, the pipeline controller kills the instructions in the earlier stages. Another function of this unit is stalling the pipeline when a stall signal is received from the ICache or the DCache.

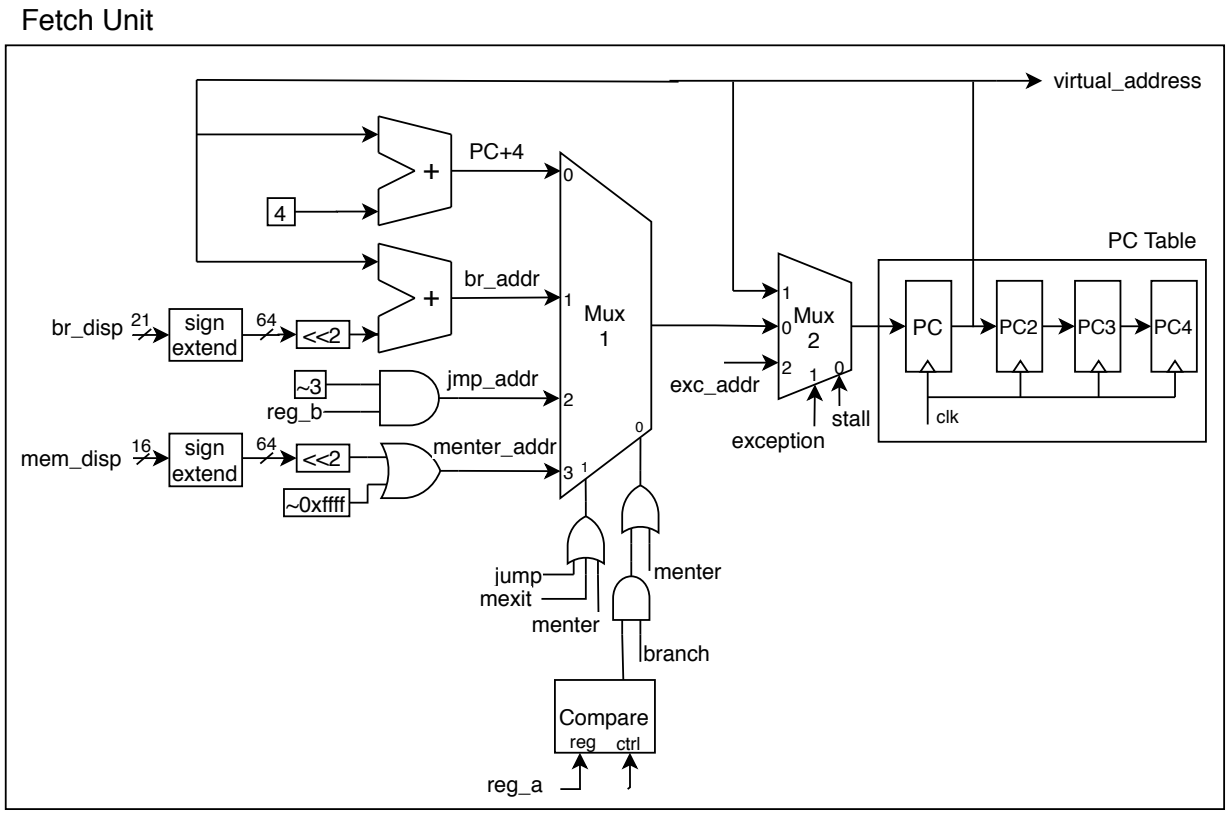

Figure 2.5: Fetch unit block diagram.

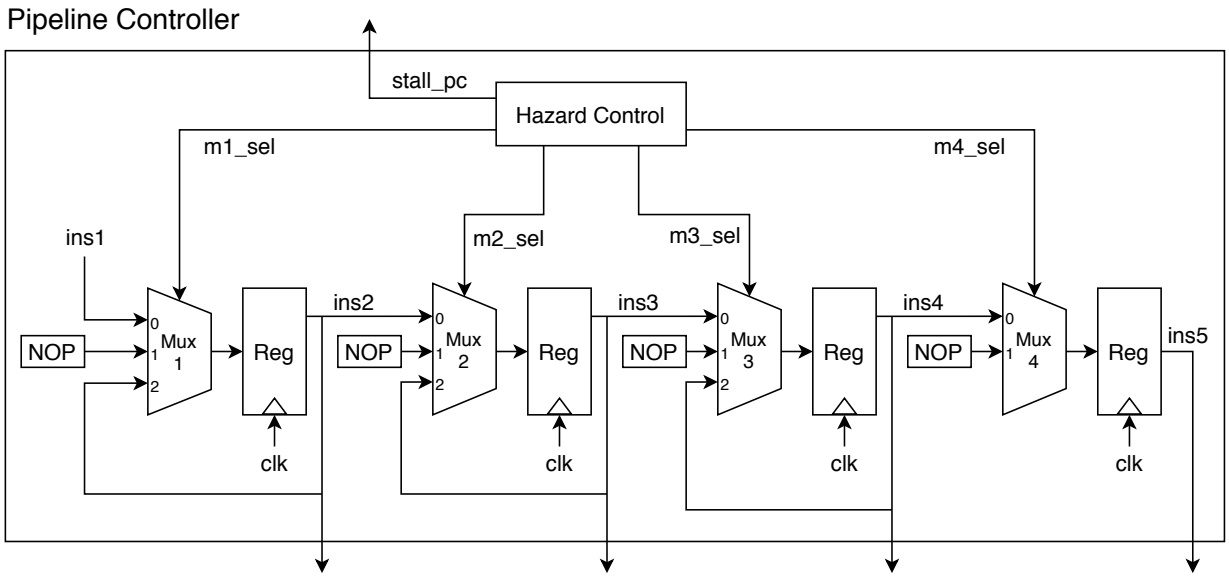

Figure 2.6: Pipeline controller block diagram.

2.6 IRF: Register File

Figure 2.7 shows the internal design of the IRF which consists of a register file and logic for delivering two operands to the IBox. The register file has two asynchronous read ports and one synchronous write port and keeps 32 64-bit registers (R0-R31). The value of R31 is always zero and any write to this register is ignored. In Figure 2.7, read addresses *read_addr1* and *read_addr2* come directly from instruction bits, but write inputs *write_addr*, *write_data*, and *write_en* are received from the MBox.

The IRF sends two 64-bit operands,A and B, to IBox. *Mux 3* and *mux 4* in Figure 2.7 select the source of these operands. In load and store instructions, A is the sign-extended displacement and B comes from the register file or forwarding paths. Operate format instructions use the register file output or bypass data as the source operands. If they contain a literal constant, B comes from the sign-extended literal. In jump or `MENTER` instructions, the updated PC value is passed to the IBox.

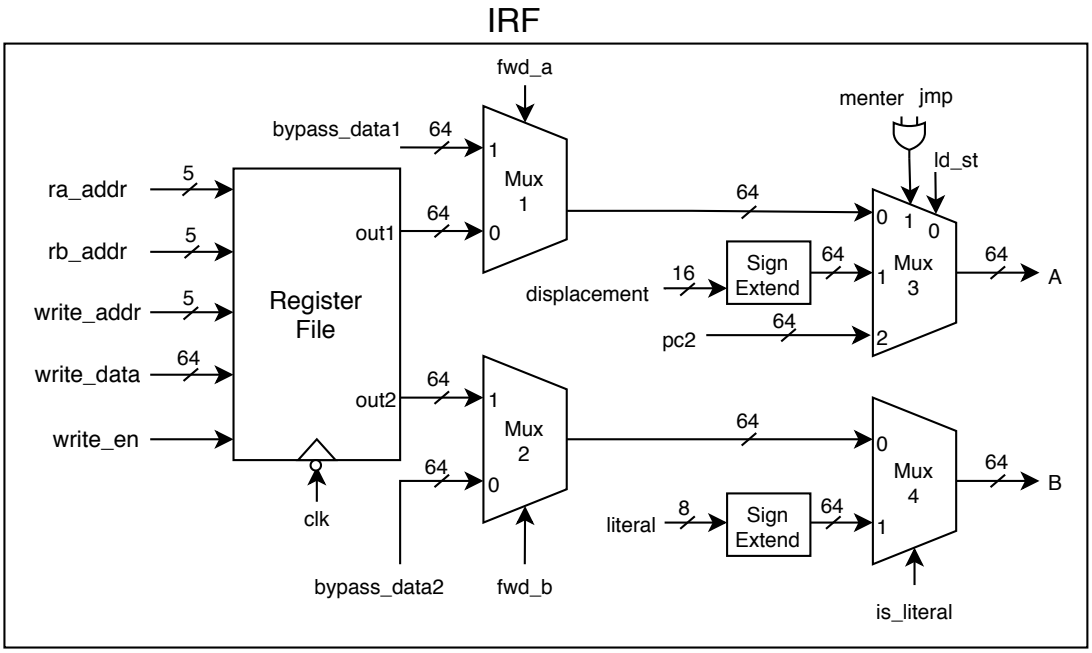

Figure 2.7: IRF block diagram.

2.6.1 Operand Forwarding

In Figure 2.7, *Mux 1* and *Mux 2* select between the register file output and the bypass data from other stages. Assume three consecutive instructions as follows:

```
i1 ra1, rb1, rc1
i2 ra2, rb2, rc2
i3 ra3, rb3, rc3
```

When `i3` is in the second stage of the pipeline, the *bypass_data1* shown in Figure 2.7 comes from:

1. The IBox result in stage 3, if `i2` is an operate format instruction and `ra3` is the same as `rc2`, also if `i2` is a jump or `MENTER` instruction and `ra3` is the same as `ra2`.
2. The IBox result in stage 4, if the above conditions exist with `i1` instead of `i2`.
3. The data read from the DCache in stage 4, if `i1` is a load instruction and `ra3` is the same as `ra1`.

4. The Metal register file output in stage 4, if `i1` is a `MRPCR` instruction and `ra3` is the same as `ra1`.

5. The value of `rb2` in stage 3, if `i2` is a conditional move instruction and `ra3` is the same as `rc2`.

6. The value of `rb2` in stage 4, if `i1` is a conditional move instruction and `ra3` is the same as `rc1`.

The *bypass_data2* is generated the same way, but `rb3` is checked instead of `ra3`.

The above list includes all data hazard cases except for one. If `i2` is a load or `MRPCR` instruction and `ra3` or `rb3` are the same as `ra2`, then the register value that `i3` needs in stage 2 is not ready in any stages of the pipeline. In this case, the pipeline controller adds a NOP instruction between `i2` and `i3`. After that, bypass data is set in a similar fashion to cases 3 and 4 of the above list.

2.7 IBox: Integer Operations

IBox is the Arithmetic and Logic Unit (ALU) that executes 64-bit integer operations in one cycle. Figure 2.8 shows the IBox diagram which contains the following units:

- U1 that consists of a 64-bit barrel shifter, an adder, a logic unit that performs AND, OR and XOR operations, and a compare unit (Figure 2.9).
- U2 that is a multiplier used for longword multiply (MULL), quadword multiply (MULQ) and unsigned quadword multiply high (UMULH) instructions.
- U3 that is used for the following instructions:
  - Pixel error (PERR)
  - Count leading zero (CTLZ), Count population (CTPOP), Count trailing zero (CTTZ)
  - Pack bytes (PKxB), and unpack bytes (UNPKBx)
- U4 that executes mask (MSKxx), insert (INSxx), extract (EXTxx), and zero bytes (ZAPx) instructions (Figure 2.10).

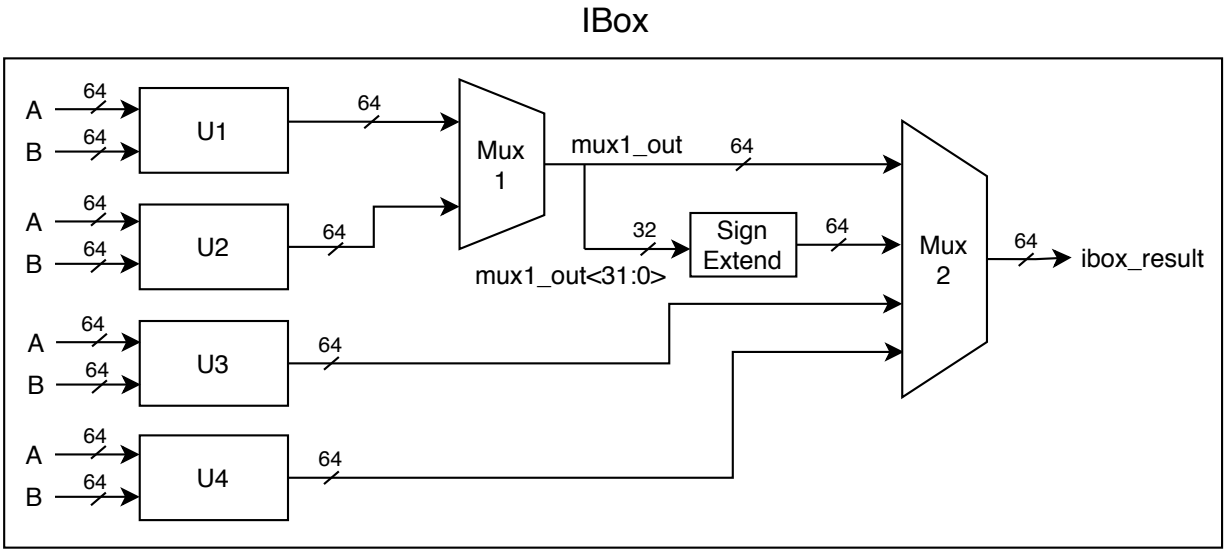

Figure 2.8: IBox block diagram.

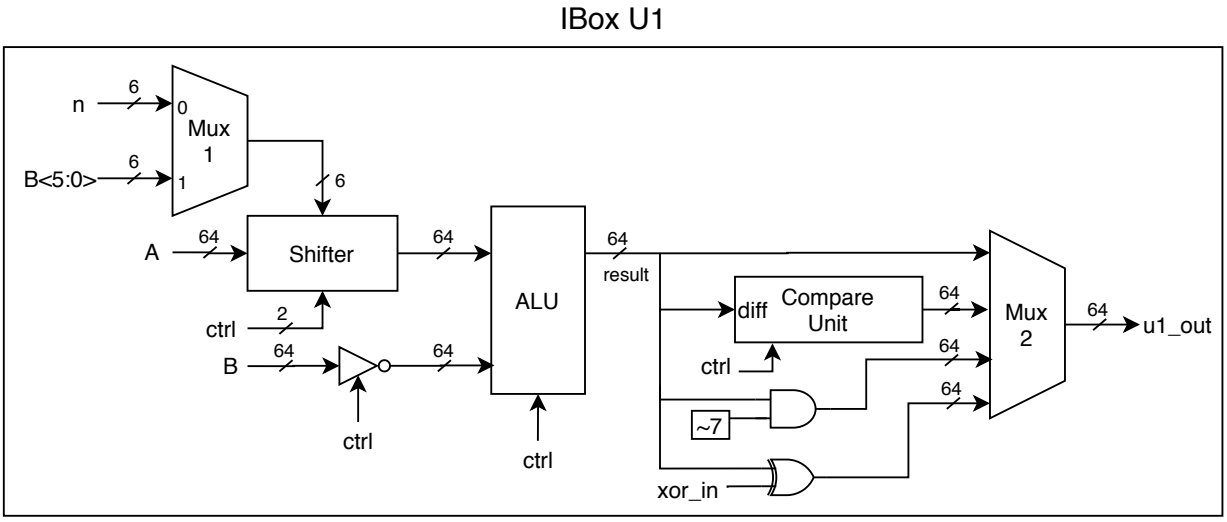

Figure 2.9: IBox U1 block diagram.

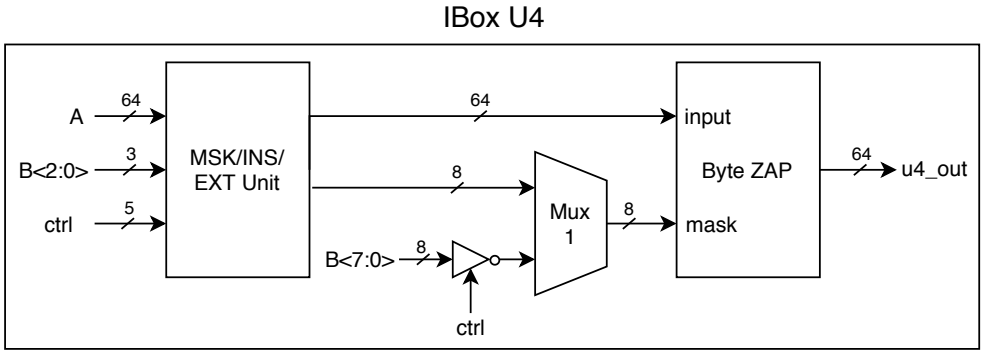

Figure 2.10: IBox U4 block diagram.

2.8 MBox: Memory Operations

MBox is the memory reference unit that performs load/store, Metal load/store (MLD/MST), and Metal read/write processor control register (MRPCR/MWPCR) instructions. As shown in Figure 2.11, the MBox is made up of the following components.

- DCache control unit: interfaces to the data cache, sign-extends or zero-extends the memory value, and keeps a locked flag to implement load-locked (LDx_L) and store-conditional (STx_C) instructions
- Metal registers: A 16-entry register file that can be accessed with MWPCR and MRPCR instructions when the processor is running in Metal mode
- Register write-back unit: Prepares the data and the address for writing in the registers. The write data can be the result of an integer operation, the memory value for a load instruction, the updated PC for a jump or MENTER, or the register value for a conditional move or MRPCR.

2.9 Cache

The cache hierarchy in our implementation is shown in Figure 2.12. We use three levels of 8-way set-associative caches with the specifications summarized in Table 2.1. For simplicity of the cache coherence protocol, L1 and L2 caches use the write-through policy, and L3 which interfaces with the memory uses the write-back policy.

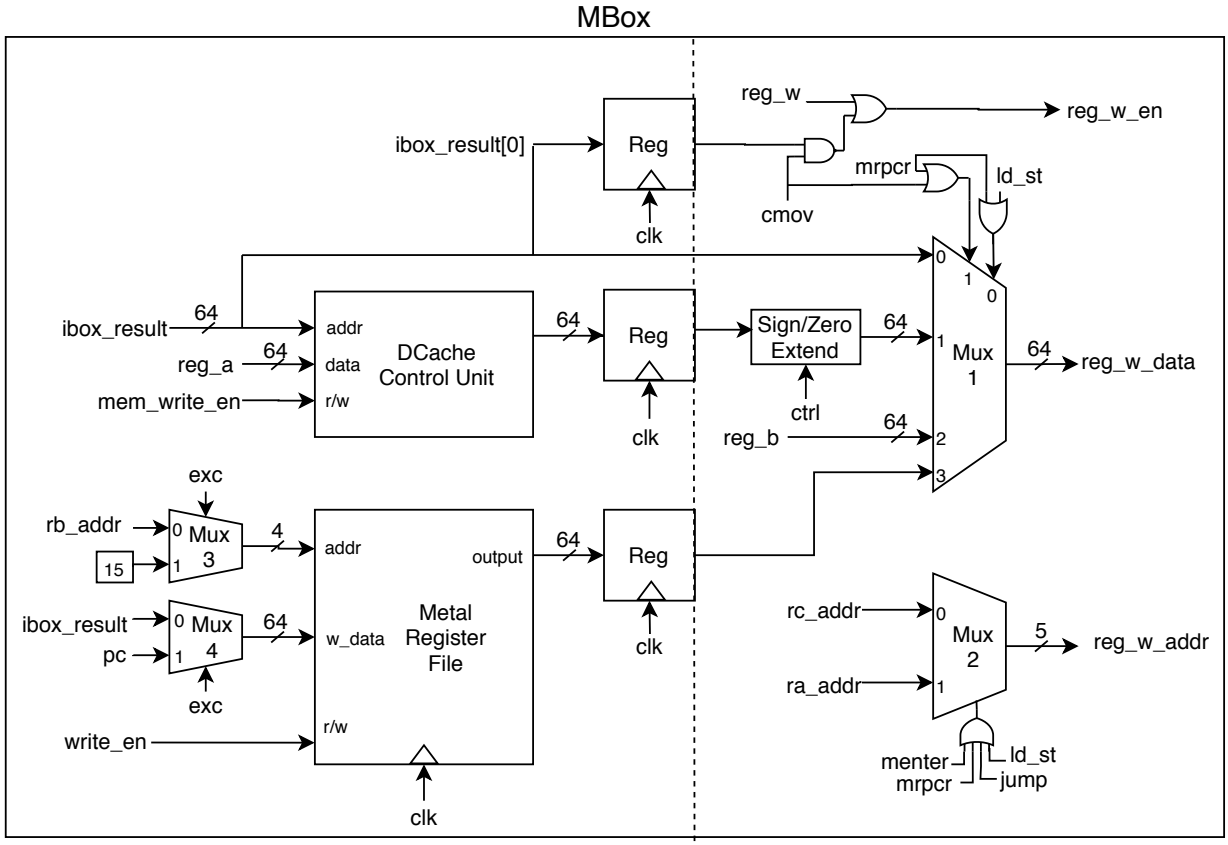

Figure 2.11: MBox block diagram.

Table 2.1: Specifications of different levels of the cache hierarchy.

Level	Associativity	Block Size	Sets	Size	Write Policy
L1-Instruction	8	64 B	64	32 KB	Write Through
L1-Data	8	64 B	64	32 KB	Write Through
L2	8	64 B	512	256 KB	Write Through
L3	8	64 B	4096	2048 KB	Write Back

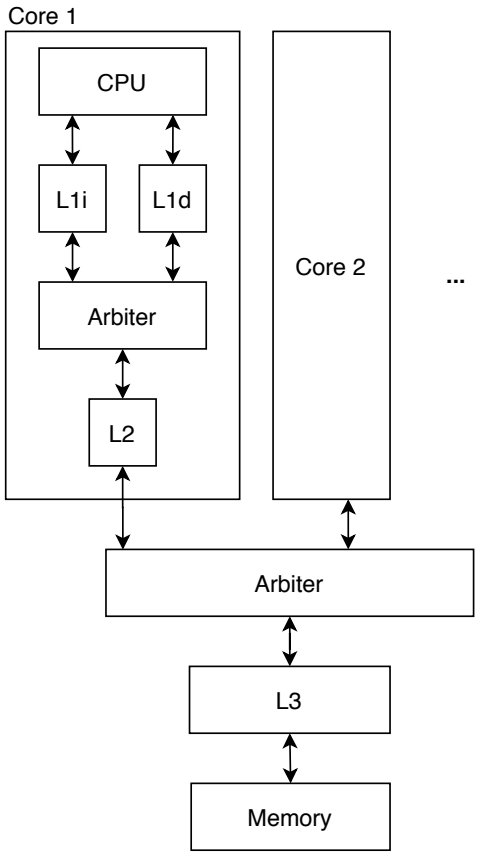

Figure 2.12: Cache hierarchy.

Table 2.2: PLRU lookup and state transitions (x means do not care, _ means unchanged).

Current State	PLRU Index
00x0xxxs	0
00x1xxx	1
01xx0xx	2
01xx1xx	3
1x0xx0x	4
1x0xx1x	5
1x1xxx0	6
1x1xxx1	7

Referenced Block	Next State
0	11_1___
1	11_0___
2	10__1__
3	10__0__
4	0_1__1_
5	0_1__0_
6	0_0___1
7	0_0___0

2.9.1 Replacement Policy

The replacement policy in all caches is tree-based pseudo-LRU. Tree-PLRU is an efficient and practical method of implementing the Least Recently Used (LRU) policy which uses a binary search tree to keep track of recently accessed cache blocks in each set. Each node of the tree has a binary flag showing whether the PLRU block resides in the left or right subtree of the current node. To find the PLRU block, the tree should be traversed from the root according to the flags. And, upon accessing a block, the binary flags on the path from that block to the root are flipped.

In an 8-way set-associative cache the PLRU tree state can be represented in 7 bits. Table 2.2 shows the mapping from the current state to the PLRU block and also the state transition when a block is referenced. For example, if the cache receives a write request to an address located in block 0 of a set, according to Table 2.2, the first, second, and fourth bits of its current state change to 1. Another example is when a cache miss occurs, and one of the blocks in a set needs to be replaced with the new data. For instance, if the current state of the matching set is 1111000, then block 6 is evicted and replaced. When the new block is read, the PLRU state changes to 0101001.

2.9.2 Arbiter

We design a bus arbiter to connect two or more upper-level caches to a lower level cache. Figure 2.13 shows the arbiter ports and connections to two levels of caches. The state

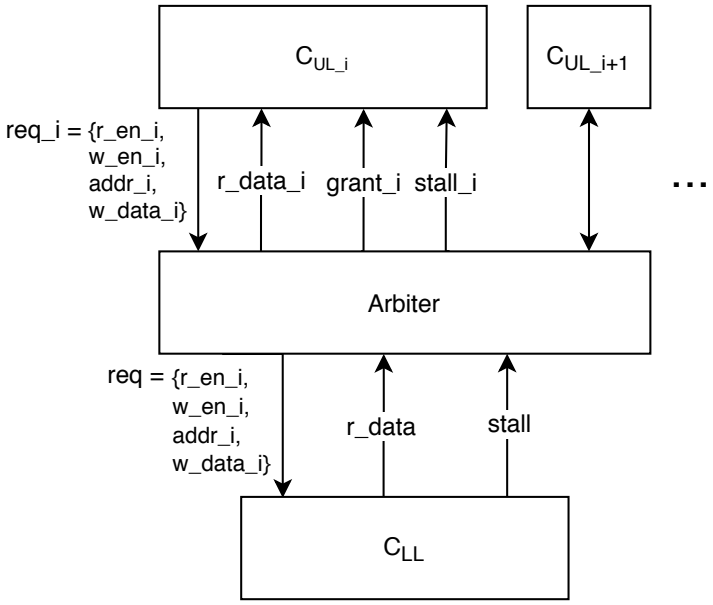

Figure 2.13: Arbiter interface.

machine for the arbiter is demonstrated in Figure 2.14. It forwards requests and answers using a round-robin scheme.

2.9.3 Cache Implementation

Figure 2.15 shows the cache interface to upper and lower-level resources. The upper level is the CPU for L1 and arbiter for other caches. The lower level is the memory for L3 and arbiter for others. The *inv*, and *inv_addr* ports only exist in L1 and L2 caches, and *st_c* only exists in L1.

The cache design includes a data store and two modules that are shown in Figure 2.16. The data store contains a number of sets indexed by *idx* bits of the address. Each set keeps several blocks and an LRU state. Inside a block, *tag*, *v* (valid), *d* (dirty), and *data* fields are stored. The tag finder module searches for a specific block based on *tag* and *idx* inputs, if it exists inside the cache and is valid, the *hit* signal is set and *bid* gives the block number. The PLRU8 unit shown in Figure 2.16 implements the replacement policy as described in section 2.9.1. This module gets the LRU state of a set and the referenced block number and produces the next state and also the LRU block number. The tag finder and PLRU8 have combinational logic and do not change any states.

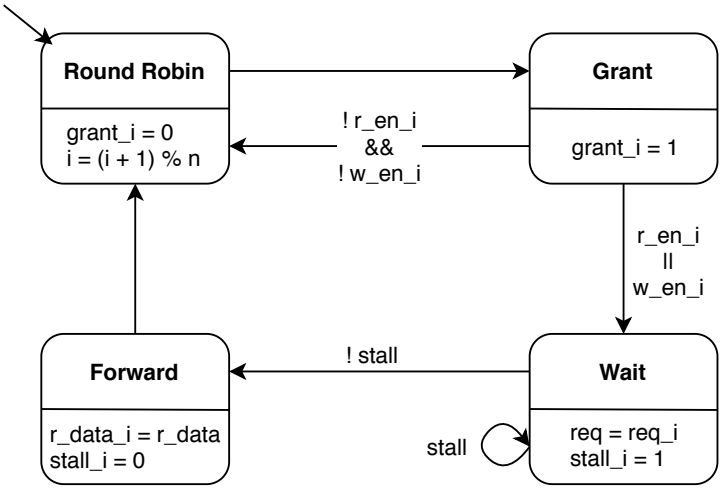

Figure 2.14: State machine for the bus arbiter.

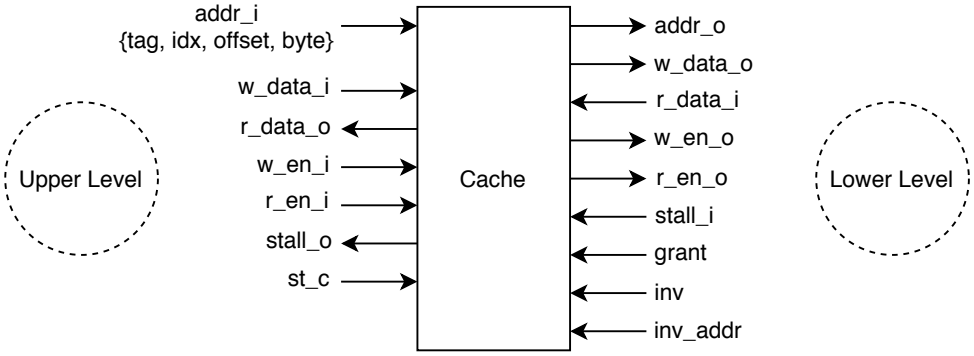

Figure 2.15: Cache interface.

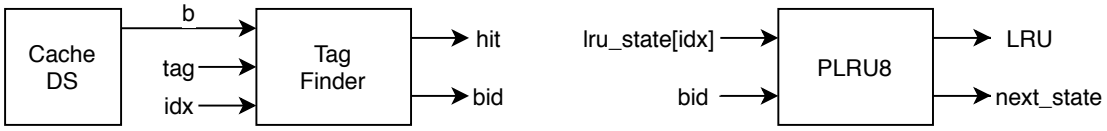

Figure 2.16: Tag finder and PLRU8 units inside a cache.

The state machine of cache is illustrated in Figure 2.17. *WT* is a parameter, not an input signal, which is only used at the module generation time and determines whether the cache is write-through. Since L3 is write-back, it does not have *Write_Through*, *WT_Granted*, and *WT_Done* states. Both write-through and write-back caches use write allocate approach when a miss occurs. Conditional stores are handled by L1, therefore, *Fail* and *Conditional_Store* states only exist in L1. Figure 2.17 summarizes the overall operation of the cache except for cache coherency which is explained in the next section.

2.9.4 Cache Coherence

We use bus snooping for maintaining cache coherency. The *w_en_i* and *w_addr_i* signals of the L3 cache are connected to *inv* and *inv_addr* of L2 and L1 caches respectively. The snooping unit works in parallel with the main operation of the cache. A separate tag finder module is used to find *inv_addr* in its corresponding set. If *inv* is high and *inv_addr* exists in the cache (and the current cache is not the one writing to this address), the valid bit of matching block is set to 0. Additionally, if the cache receives an invalidation in the middle of processing a read or a write to the same address, in L2 caches, the current state immediately changes to *Idle* and L1 caches go to the *Search* state. In this case, the request is retried from the top by L1.

2.9.5 Lock Mechanism

Load locked and store conditional instructions are used in pairs to perform atomic memory updates. The processor keeps a lock flag that is set by LDx_L instructions. The succeeding STx_C instruction only executes if the lock flag is still set. The lock flag is cleared after a conditional store or a context switch. In this case, if the program is interrupted between an LDx_L/STx_C pair, the store fails. To guarantee that the locked data is not modified by another core, the L1 only executes a conditional store on a cache hit. A successful STx_C comes after an LDx_L without any other memory references in between, therefore, the requested address should be available on the cache. If a miss occurs, it means that another core might have written to the same address and the data is invalidated. As shown is Figure 2.17, *r_data_o* is 1 when conditional store succeeds and 0 when it fails.

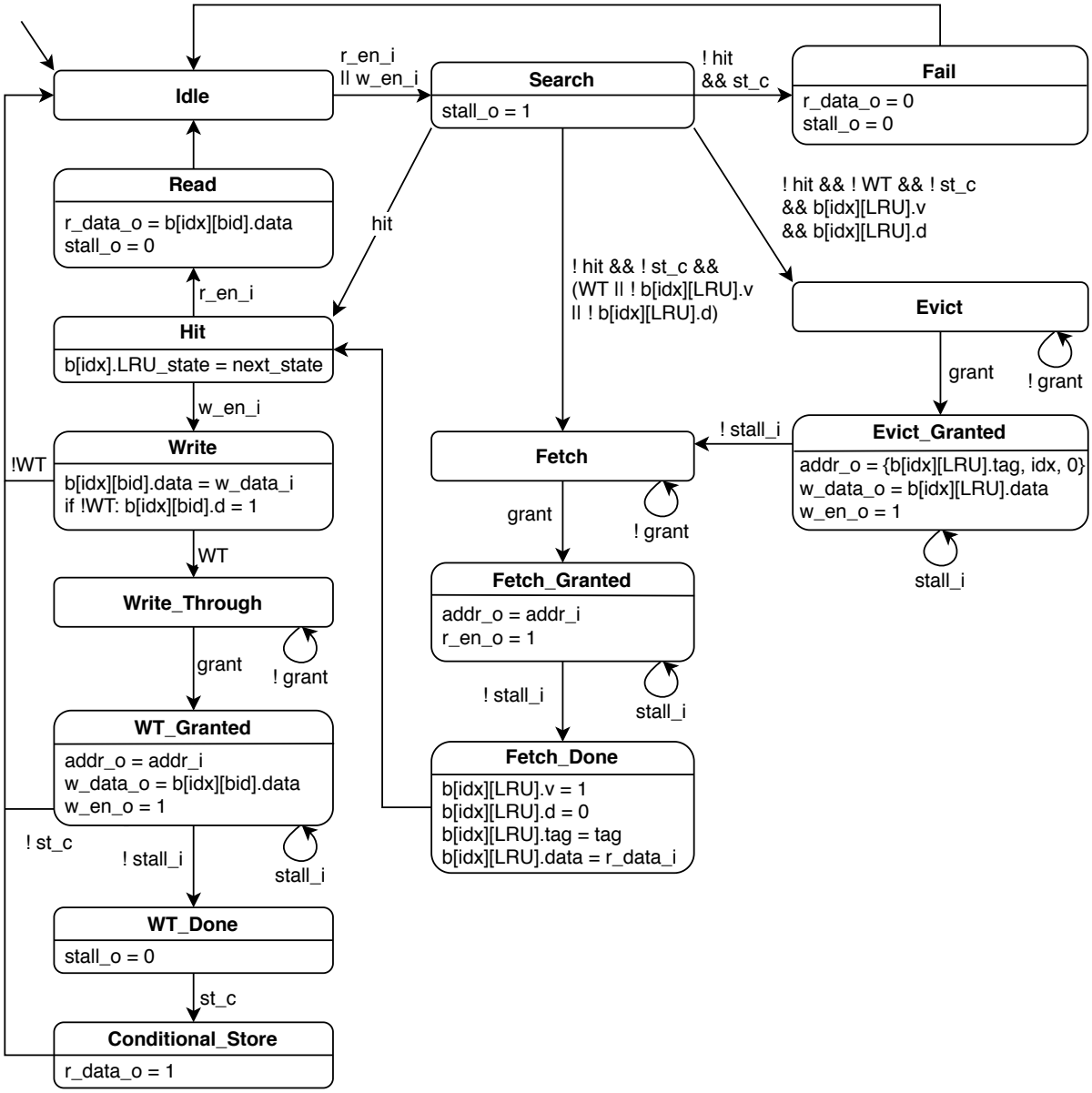

Figure 2.17: State machine for the cache. The inputs and outputs shown in this state machine are based on the signals shown in Figure 2.15 and Figure 2.16 (WT is a module parameter, not an actual input).

Chapter 3

Metal: An Alternative Privileged Architecture

3.1 Overview

The privileged architecture for many systems consists of functions implemented in hardware and made available to the operating system through instruction set extensions. This means changing the existing mechanisms or creating new designs is impossible for operating system developers. We propose Metal, a new abstraction that allows the privileged architecture to evolve through software rather than hardware. In the following sections, we describe the Metal design and implementation.

3.2 Metal Design

3.2.1 Design Overview

The Metal architecture consists of a special operating mode, six new instructions, and some processor control registers (PCR). It replaces the privileged architecture of conventional systems. We consider the following design goals for Metal:

- Instantaneous control transfers to Metal mode
- Interpose on the operating system, hypervisors, and applications

- Secure from the system software component

Only two operating modes exist in Metal, normal mode, and Metal mode. Transitions between these two modes are available through two new Metal instructions. All architectural features of the processor are controlled in Metal mode through the addressable processor control register space.

Any number of privilege levels can easily be implemented with Metal. Changing the privilege mode is done by a Metal call. The processor enters the Metal mode and then checks the legality of the transition between the two privilege modes. If the transition is allowed, it properly changes the state in PCR and then exits to normal mode and returns control to the target. Application, kernel, or hypervisor code all run in normal operation mode.

3.2.2 Design

The Metal architecture consists of Metal memory, registers, and instructions which can be added to any microarchitecture. Metal code is written in standard machine code with the addition of Metal instructions and is kept in a microcode-like RAM for fast access. The six new instructions used by Metal are listed in Table 3.1. MENTER instruction changes the operation mode from normal to Metal and then jumps to an address in Metal memory. In contrast, MEXIT is used for leaving the Metal mode. It performs like a regular jump to a given location in the instruction address space. MWPCR and MRPCR are used to move a value from general-purpose registers to Metal registers and vice versa. MLD and MST bypass the memory management hardware and access the physical memory directly. All Metal instructions except for MENTER can only be executed in Metal mode.

All architectural features are controlled through the addressable processor control register space. Architectural features include basic CPU information, virtual memory, performance counters, and registers reserved for Metal.

3.3 Metal Implementation

This section describes our implementation of Metal for Alpha architecture in μAlpha.

Instruction	Description
MENTER	Enter Metal Mode
MEXIT	Exit Metal Mode
MRPCR	Metal Read Processor Control Register
MWPCR	Metal Write Processor Control Register
MLD	Metal Load from Physical Memory
MST	Metal Store to Physical Memory

Table 3.1: Instructions for the Metal architecture.

3.3.1 Metal Memory

The last page (64 KB) of the address space is assigned to Metal. If the PC value falls in this range, the instruction is fetched from the Metal RAM in pipeline stage 1. The Metal RAM resembles the microcode storage in microcode machines. Metal programs are loaded in this RAM for low latency control transfers.

The first 64 words of the RAM act as a table for Metal subroutines. They are the entry points for Metal calls and can be filled with jumps to different subroutines. We explain more about Metal calls in section 3.3.3.

3.3.2 Metal Registers

We use a 32-entry register file as the processor control registers, MR0 to MR31. Similar to load and stores to the main memory, this register file is accessed in stage 4 of the pipeline in order to move a value from general registers to Metal registers or vice versa. These registers can be used for any purpose in a Metal code, for example, to keep the user/kernel status.

3.3.3 Metal Instructions

We use the branch format for MENTER and MEXIT instructions and the memory format for the other Metal instructions. Table 3.2 lists the format and operation of each Metal instruction.

MENTER saves the old PC in RA and then updates it with a Metal address based on the number specified in the displacement field. For example `MENTER R26,#1` jumps to

Instruction	Format	Operation

`MENTER Ra, disp`	Branch format	`{update PC}`
		`Ra <- PC`
		`va <- {4 * ZEXT(disp)} AND 0x3f`
		`PC <- va OR 0xfffffffffff0000`
		`PS <- 1`

`MEXIT Ra, disp`	Branch format	`PS <- 0` `PC <- Ra`
`MRPCR Ra, MRb, disp`	Memory format	`IF PS == 1 THEN` `  Ra <- MRb`
`MWPCR Ra, MRb, disp`	Memory format	`IF PS == 1 THEN` `  MRb <- Ra`

`MLD Ra, Rb, disp`	Memory format	`IF PS == 1 THEN`
		`  pa <- Rb + SEXT(disp)`
		`  Ra <- (pa)`

`MST Ra, Rb, disp`	Memory format	`IF PS == 1 THEN`
		`  pa <- Rb + SEXT(disp)`
		`  (pa) <- Ra`

Table 3.2: The format and operation of the Metal instructions.

the second instruction in Metal RAM which has the address 0xfffffffffff0001, and saves the return address in R26.

As mentioned in the previous sections, there are two processor modes, Metal and normal. The current mode is stored in the Processor Status mode bit (PS). MENTER enters the Metal mode and MEXIT leaves it by changing the PS. MRPCR, MWPCR, MLD, and MST instructions only execute if the PS is set to Metal mode.

3.3.4 Security

The Metal code and data are isolated from the operating system, so the OS can not tamper with them directly. In addition to separated instruction RAM and register file, Metal keeps its data in a small portion of the main memory which is hidden from the system software and can only be accessed with MLD and MST instructions.

MENTER can only jump to the first 256 bytes of the Metal RAM. This prevents the programs from bypassing the security checks by jumping in the middle of Metal subroutines.

Even if a normal jump is used for entering the Metal code, the processor mode has not been changed to Metal so Metal instructions (MWPCR, MRPCR, MLD, and MST) can not be executed and Metal data remains secure.

3.3.5 Exception Handling with Metal

When executing an instruction generates an exception, the following steps are taken:

- The address of the instruction causing the exception is saved in one of the processor control registers. Metal register 15 is reserved for this purpose.
- Instructions in the earlier stages of the pipeline are killed.
- The processor status bit (PS) is set to Metal mode.
- The PC is loaded with the address of the Metal subroutine for handling exceptions.

Chapter 4

Metal Applications and Evaluation

This chapter describes our implementation of system calls and transactional memory as two examples of Metal application. Metal can be used in many different areas to improve the security and reliability of future operating systems. We discuss some other applications and benefits of Metal at the end of this chapter. Finally, we provide a qualitative evaluation of the Metal programming model.

4.1 System Calls

In this section, we describe a system call example that we implemented using Metal. Figure 4.1 shows the mechanism of a system call for a UNIX-like operating system. Privilege levels, in this case, user and kernel, are defined by Metal. We use one of the Metal registers (MR0) to keep the privilege mode. This is different from the processor status register which keeps the Metal/normal mode. Two Metal subroutines, `syscall` and `sysreturn`, are defined for transitions between the privilege levels. The `syscall` code is as follows:

```
1  # Input: Syscall number in R16
2  SyscallEntry:
3      MWPCR R31, MR0, 0           # Set to kernel mode
4      AND R16, 0x3f, R1           # Syscall number range
5      SLL R1, 3, R1
6      LDQ R1, R1, SYS_TABLE_ADDR  # Read syscall table
7      MEXIT R1, 0                 # Jump to syscall code
```

Listing 4.1: Syscall implementation in Metal.

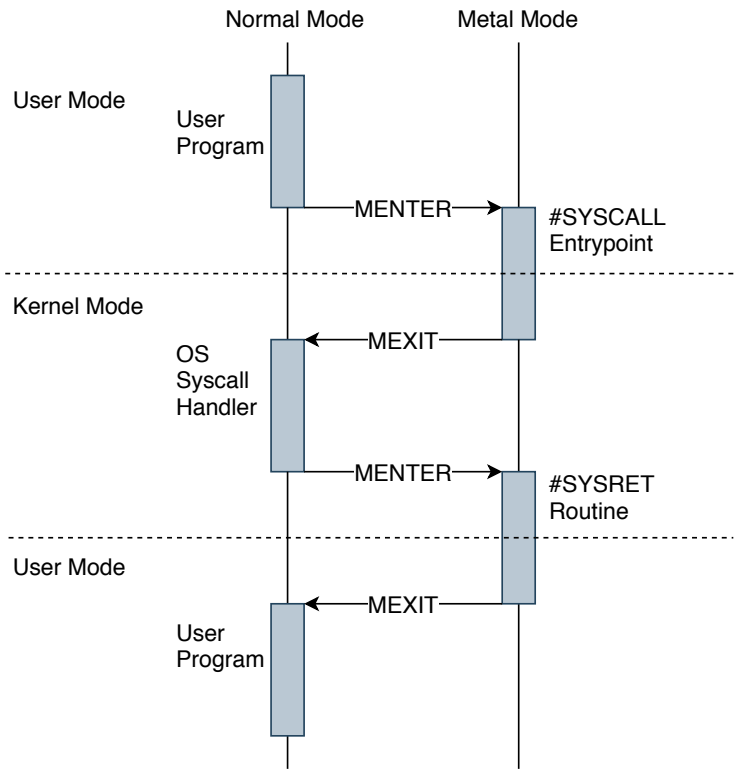

Figure 4.1: A typical Metal call. The mechanism's strength lies in its simplicity and generality.

First, the code sets the mode register (MR0) to 0, which means the kernel mode, and then looks up the syscall number in the syscall table and jumps to the obtained address. By using MEXIT to jump to the syscall code, we leave the Metal mode but remain in kernel mode. In order to execute a system call, The user needs to put the system call number in an argument register (R16) and then use MENTER with the `syscall` subroutine number, which is 1 in our implementation. The return address should be saved in R26. The return address and other possible arguments are passed to the OS syscall code. The following example shows calling system call number 1 with an argument set to 0:

```
1  ADDQ R31, 0x1, R16
2  ADDQ R31, R31, R17
3  MENTER  R26, 1
```

Listing 4.2: Example of using a system call.

Every system call code starts with saving the return address and ends with calling the `sysreturn` subroutine and passing the return address to it. The following code shows a system call example:

```
1  ADDQ R31, R26, R9   # Back up return address
2  ADDQ R31, R31, R16
3  MENTER R26, 3       # Read the protected data (MR1) [Output: R27]
4  ADDQ R27, 1, R17    # Increment by 1
5  ADDQ R31, 1, R16
6  MENTER R26, 3       # Write the protected data [Input: R17]
7  ADDQ R31, R9, R26   # Restore return address
8  MENTER R31, 2       # Sysreturn
```

Listing 4.3: Example of a system call.

At line 8 of the above code, MENTER is used with R31 (which is ignored as a destination register) because the next instruction address is not needed and the execution never returns to this point. The following code shows the `sysreturn` subroutine:

```
1  # Input: Return address in R26
2  SysreturnEntry:
3      ADDQ R31, 1, R1
4      MWPCR R1, MR0, 0    # Set to user mode
5      MEXIT R26, 0        # Return to user code
```

Listing 4.4: Sysreturn implementation in Metal.

The privileged reads from and writes to a protected memory location or register are performed by Metal subroutines in which the current mode is checked before accessing the secured data. An exception is generated if these Metal subroutines are called outside of a system call where the code is running in user mode. In the above example, Metal subroutine number 3 is called for reading and writing to MR1 which contains a value that can only be modified in kernel mode. This subroutine is defined as follows:

```
1  # R16: Read or write, R17: write data, R27: read data
2      MRPCR R1, MR0, 0
3      BNE R1, 5          # Check kernel mode
4      BEQ R16, 2
5      MWPCR R17, MR1, 0  # Write
6      MEXIT R26, 0
7      MRPCR R27, MR1, 0  # Read
8      MEXIT R26, 0
9      MWPCR R26, MR15, 0 # Exception
10     ADDQ R31, R31, R16
11     MENTER R31, 0      # Jump to exception handler
```

Listing 4.5: Metal subroutine for reading and writing in a protected register.

4.2 Transactional Memory

Transactional memory is a concurrency control mechanism similar to database transactions and provides controlled access to regions of the memory that are shared between concurrent processes. Transactional memory was originally proposed to be implemented in hardware. It was first introduced by Knight [24] and then popularized by Herlihy and Moss [17]. Shavit and Touitou [32] propose a software-only implementation for the first time that can be implemented on arbitrary hardware using load-linked/store-conditional primitives.

In this section, we present a simple transactional memory (TM) interface implemented in Metal that provides controlled access to specific regions of memory that are shared between several processes. Those regions of memory are only accessible through atomic and serializable transactions and are protected against direct modifications by user-space processes. Atomicity ensures that either all changes made by a transaction are applied (in

case transaction successfully commits) or none of them are applied (in case transaction fails). Serializability ensures that changes made by transactions appear to be the result of running transactions in a specific sequential order (i.e., the steps of one transaction are never interleaved with the steps of another transaction).

The simple transactional memory that we implement requires two global variables called `Global_Lock` and `Global_Version` that should reside in a memory address that is private to Metal (not accessible from user-space). `Global_Lock` is set to 1 when a transaction is being committed and is 0 when no transaction is currently committing. `Global_Version` is a counter that keeps track of the version of the TM which is increased by one whenever a transaction is successfully committed.

Our transactional memory implementation is composed of five Metal subroutines: `TInit`, `TStart`, `TRead`, `TWrite`, and `TCommit`. `TInit` initializes the TM interface globally and should be called only once. The rest of the subroutines are specific to a single transaction. A buffer called LogStruct should be allocated by the user-space process that is starting a new transaction. This buffer will contain all information and buffers for a single transaction and needs to be passed to all TM calls related to that transaction. Also, all read and write operations are done on quad-words and physical address is used for read and write operations.

4.2.1 TInit

`TInit` is called only once before all TM operations. It initializes the `Global_Lock` and `Global_Version` to 0 to enable the correct functionality of other subroutines. Code 4.6 shows `TInit` subroutine.

```
1  TInitEntry:
2      MST R31, R31, VERSION_ADDR  # Reset Global_Version to 0
3      MST R31, R31, LOCK_ADDR     # Reset Global_Lock to 0
4      MEXIT R26, 0                # Return
```

Listing 4.6: TInit implementation in Metal (no inputs and outputs).

4.2.2 TStart

`TStart` is called to start a new transaction. It will initialize the LogStruct buffer allocated by the user code to represent an empty transaction created at that time. The size

of the allocated LogStruct should be at least $2 + MAX\_WRITES \times 2$ quad-words with MAX_WRITES being the maximum number of writes allowed for the transaction. Figure 4.2 shows the different fields inside the LogStruct buffer. `TStart` sets the "Write_Buffer Length" field to 0 and copies the current `Global_Version` to the `Start_Version` field. `Start_Version` is examined at commit time to check if another transaction has been committed since this transaction has begun. Write buffer keeps track of all modifications made in the transaction by storing (Written Address, Written Value) pairs.

Start_Version	Write_Buffer Length	Write_Buffer Address_1	Write_Buffer Value_1	...	Write_Buffer Address_n	Write_Buffer Value_n

Figure 4.2: Structure of the LogStruct buffer for each transaction.

```
1  TStartEntry:
2      MLD R1, R31, VERSION_ADDR   # Read Global_Version
3      STQ R1, R16, 0              # LogStruct.Start_Version = Global_Version
4      STQ R31, R16, 8             # LogStruct.Write_Buffer_Length = 0
5      MEXIT R26, 0                # Return
```

Listing 4.7: TStart implementation in Metal (Input: LogStruct address at R16, Output: None).

4.2.3 TRead

`TRead` is used to perform a read operation in a transaction. `TRead` scans the write buffer in the LogStruct to see if the read address is previously written in the current transaction. If it is written, it returns the modified value from the write buffer. Otherwise, it returns the value directly read from the physical address.

```
1  TReadEntry:
2      LDQ R1, R16, 8            # R1 = LogStruct.Write_Buffer_Length
3      ADDQ R31, R31, R2         # R2 = 0 (Loop Iterator)
4  loop:
5      CMPLT R2, R1, R3.         # R3 = R2 < R1 (Loop Condition)
6      BEQ R3, +9 (loop_end)     # Break loop if R2 >= R1 (Buffer Length)
7      SLL R2, 4, R3             # R3 = R2 x 16 (2 words)
8      ADDQ R16, R3, R3          # R3 += LogStruct address
```

```
 9      LDQ R4, R3, 16            # R4 = LogStruct.Write_Buffer[R2].address
10      CMPEQ R4, R17, R5         # R5 = R4 == read_address
11      BEQ R5, +2 (continue)     # Go to next iteration if R5 is False
12      LDQ R27, R3, 24           # Result = LogStruct.Write_Buffer[R2].value
13      BR R31, +3 (exit)         # Break Loop and Exit
14  continue:
15      ADDQ R2, 1, R2  # Increment R2 (Loop Iterator)
16      BR R31, -11 (loop_start)  # Loop
17  loop_end:
18      MLD R27, R17, 0 # Read the actual physical address
19  exit:
20      MEXIT R26, 0        # Return
```

Listing 4.8: TRead implementation in Metal (Input: LogStruct address at R16, Read physical address at R17, Output: Read value in R27).

4.2.4 TWrite

`TWrite` is used to perform a write operation in a transaction. `TWrite` scans the write buffer in the LogStruct to see if the write address is previously written in the current transaction. If it is written, it updates its value in the write buffer. Otherwise, it increases the length of the write buffer by one and adds the new (Address, Value) pair to the write buffer.

```
 1  TWriteEntry:
 2      LDQ R1, R16, 8           # R1 = LogStruct.Write_Buffer_Length
 3      ADDQ R31, R31, R4        # R2 = 0 (Loop Iterator)
 4  loop:
 5      CMPLT R2, R1, R3   # R3 = R2 < R1 (Loop Condition)
 6      BEQ R3, +9 (loop_end)    # Break loop if R2 >= R1 (Buffer Length)
 7      SLL R2, 4, R3            # R3 = R2 x 16 (2 words)
 8      ADDQ R16, R3, R3         # R3 += LogStruct address
 9      LDQ R4, R3, 16           # R4 = LogStruct.Write_Buffer[R2].address
10      CMPEQ R4, R17, R5        # R5 = R4 == write_address
11      BEQ R5, +2 (continue)    # Go to next iteration if R5 is False
12      STQ R18, R3, 24          # LogStruct.Write_Buffer[R2].value = R18
13      BR R31, +8 (exit)        # Break Loop and Exit
14  continue:
15      ADDQ R4, 1, R4 # Increment R2 (Loop Iterator)
16      BR R31, -11 (loop)       # Loop
17  loop_end:
18      SLL R1, 4, R2  # R2 = Buffer Length (R1) x 16 (2 words)
```

```
19      ADDQ R16, R2, R2          # R2 += LogStruct address
20      STQ R17, R2, 16           # LogStruct.Write_Buffer[R1].address = R17
21      STQ R18, R2, 24           # LogStruct.Write_Buffer[R1].value = R18
22      ADDQ R1, 1, R1            # Increment buffer length by 1
23      STQ R1, R16, 8            # LogStruct.Write_Buffer_Length = R1
24  exit:
25      MEXIT R26, 0        # Return
```

Listing 4.9: TWrite implementation in Metal (Input: LogStruct address at R16, Write physical address at R17, Write value at R18, Output: None).

4.2.5 TCommit

`TCommit` is used to commit a transaction. First, it attempts to acquire the `Global_Lock` by performing a compare and swap operation (implemented using load-linked/store-conditional primitives). If acquiring the lock is unsuccessful, it terminates and returns 0. If the lock is acquired, it compares `Global_Version` to `Start_Version` stored in the LogStruct. If the version does not match, it means that another transaction has committed since the current transaction has begun. So, it releases the lock and returns 0. Otherwise, the current transaction is successful and the write buffer should be flushed to the memory. In this case, `TCommit` iterates over the (Address, Value) pairs in the write buffer and writes the updated values to modified addresses. Then, it increments `Global_Version` by 1, releases the `Global_Lock` and returns 1 to show a success.

Line 1 to Line 8 implement a compare and swap mechanism using the load-linked/store-conditional primitives. Store conditional stores the new value only if the address is not modified since the previous load-linked operation on the same address. Otherwise, it will not perform the store and overwrites the source register with 0.

```
1   TCommitEntry:
2       LDQ_L R27, R31, 0x1008        # R27 = Global_Lock (load-linked)
3       CMPEQ R27, 0, R1              # R1 = Lock == 0
4       BNE R1, +2 (acquire)          # If R1 is True, acquire the lock
5       ADDQ R31, 0, R27              # Otherwise, Result = 0
6       BR R31, +22 (exit)            # Exit
7   acquire:
8       ADDQ R31, 1, R27              # R27 = 1
9       STQ_C R27, R31, 0x1008        # Global_Lock = 1 (store-conditional)
10      BEQ R27, +19 (exit)           # If store conditional fails, exit
11      LDQ R1, R31, 0x1000           # R1 = Global_Version
```

```
12      LDQ R2, R16, 0                  # R2 = Start_Version
13      CMPEQ R1, R2, R27               # R27 = (R1 == R2)
14      BEQ R27, +14 (release)          # If R27 is False, release and exit
15      LDQ R1, R16, 8                  # R1 = LogStruct.Write_Buffer_Length
16      ADDQ R31, R31, R2               # R2 = 0 (Loop Iterator)
17  loop:
18      CMPLT R2, R1, R3                # R3 = R2 < R1
19      BEQ R3, +7 (loop_end)           # Break loop if R2 >= R1 (Buffer Length)
20      SLL R2, 4, R3                   # R3 = R2 x 16 (2 words)
21      ADDQ R16, R3, R3                # R3 += LogStruct address
22      LDQ R4, R3, 16                  # R4 = Write Address
23      LDQ R5, R3, 24                  # R5 = Updated Value
24      STQ R5, R4, 0                   # Store R5 in *R4
25      ADDQ R2, 1, R2                  # Increment R2 (Loop Iterator)
26      BR R31, -9 (loop)               # Loop
27  loop_end:
28      LDQ R1, R31, 0x1000             # R1 = Global_Version
29      ADDQ R1, 1, R1                  # Increment R1
30      STQ R1, R31, 0x1000             # Global_Version += 1
31  release:
32      STQ R31, R31, 0x1008            # Release Lock
33  exit:
34      MEXIT R26, 0                    # Return
```

Listing 4.10: TCommit implementation in Metal (Input: LogStruct address at R16, Output: Commit (1) or Fail (0) in R27).

4.3 Other Applications

Metal is a powerful tool that enables the operating system developers to create new designs and abstractions. It can be used to improve microkernel design and define flexible ring models and inter-process communication (IPC) mechanisms. Many system features such as hardware capabilities, security enclaves, and virtualization that were traditionally built using microcode can be implemented with Metal.

4.3.1 Protection Rings and IPC Mechanism

Metal can be used to implement protection models beyond the basic user/kernel mode abstraction, such as ring models akin to VMS [13] with four rings and Multics [30] with

seven rings. Metal also enables designs that are not strictly hierarchical. The Metal code can easily enforce a flexible set of rules for various control transfers between rings.

Microkernels can use Metal to support fast transitions between a group of system services. Metal allows fast remote procedure calls (RPCs) into system services without requiring a full context switch. Applications or system services can quickly change permissions using the TLB page keys (PKEYs). This is similar to the use of memory segmentation in L3 and L4 microkernels to support fast RPCs [27].

4.3.2 Capability-based Security

Metal can enable a capability operating system through mechanisms similar to that of prior systems. Metal can protect a region of physical memory to keep all of its data structures and bookkeeping protected from even the operating system kernel. In this architecture, Metal manages the memory layout and TLB hardware directly.

We can build a capability model by defining Metal subroutines for creating a domain, destroying a domain, entering a domain, and exiting from a domain. Creating a domain returns a capability that enables one to destroy or call into a domain. A domain can delegate capabilities to sub-domains through a set of calls for manipulating capabilities. This includes creating a capability associated with a domain, subsetting a capability, reading the type of capability, reading the permissions, and reading the size. Another subroutine is required for the domains to get the base pointer.

Memory capabilities represent regions of memory that will be mapped into a domain. Memory mapped IO (MMIO) works similarly. Capabilities that give privileges to special Metal routines to modify the processor are passed as arguments to the specific Metal subroutine. Metal can provide coarse grain protection while compiler and language runtime support can provide fine-grained capabilities for application code.

4.3.3 Secure Enclaves

A secure enclave provides an isolated and highly protected environment for running an application and guarantees the security of its data at run-time. Secure enclaves can protect the data even from physical attacks and root-level compromise. Many architectures have built these mechanisms using a combination of hardware and microcode. On the other hand, some research systems have shown how to implement enclaves in software. We can use these software techniques to build secure enclaves in Metal, with minimum hardware requirements.

4.3.4 Virtualization

Virtualization is a good example of a feature that can be implemented with Metal. Using Metal for virtualization reduces some complexity and issues that existed in previous systems. For example, the Alpha microarchitecture had been designed with a fixed number of rings for VMS. So, when they created the hypervisor using PALcode they were forced to use ring compression, which compromised the security of the VMS operating system [23]. Perhaps, the ring compression problem also existed in VAX hypervisor [14]. A more generic architecture, such as Metal, that has a TLB with PKEYs and ASIDs may have avoided this oversight.

4.4 Evaluation

4.4.1 Performance

Placing Metal code in RAM associated with the instruction fetch unit enables low latency entry and exit of Metal mode. This local memory inside the processor resembles the microcode ROM. It enables Metal programs to be cached in the processor's prefetch stage. Therefore it takes approximately one cycle to get in and leave Metal routines. This design provides performance very close to the traditional microcode approach for most applications.

Another design component that has a role in enhancing Metal's performance is its exclusive register file. Metal uses these registers as quick storage for Metal operations. This helps minimize the number of registers we must save when entering and exiting the Metal mode, and allows those registers to be saved to the spare Metal register file. Metal routines can operate faster because fewer memory references are required.

Metal can also be used with a TLB that implements address space identifiers (ASIDs) and protection keys (PKEYs). Using ASIDs allows Metal code to activate a subset of TLB entries to avoid a TLB flush on every context switching. PKEYs enable quick changes to page permissions without requiring expensive modifications to TLB when switching modes or privilege levels. Combining these techniques can allow for efficient paging, virtualization, and enclave applications.

4.4.2 Implementation Complexity

In this section, we evaluate the overhead of implementing Metal on a processor. To do so, we synthesize μAlpha using the Yosys [37] synthesis tool and the Synopsys [2] cell library. We compare resource utilization before and after adding Metal to the design. In both cases, we measure resource utilization after optimizations. Table 4.1 shows the number of wires, the total number of cells, and the chip area.

As can be seen, after adding Metal, the number of wires is increased by 16%, the number of cells is increased by 14%, and the chip area is increased by 17%. The majority of this is dominated by the Metal memory and Metal register file. The remaining hardware changes are minimal.

For superscaler or out-of-order processors the resource requirements will negligible compared to the total processor size. The μAlpha is a very simple five-stage pipeline version of the Alpha architecture. This leads the numbers to be a worse case scenario for most practical implementations.

Table 4.1: Comparison of μAlpha synthesized with and without Metal using the Synopsys cell library.

Metric	Without Metal	With Metal	Percentage Increase
Number of Wires	170264	197705	16.1%
Number of Cells	180546	206384	14.3%
Chip Area (μm^2)	1231906	1442709	17.1%

4.4.3 Developer Experience

wIn this section, we evaluate the overhead of implementing Metal on a processor. To do so, we synthesize μAlpha using the Yosys [37] synthesis tool and the Synopsys [2] cell library. We compare resource utilization before and after adding Metal to the design. In both cases, we measure resource utilization after optimizations. Table 4.1 shows the number of wires, the total number of cells, and the chip area.

As can be seen, after adding Metal, the number of wires is increased by 16%, the number of cells is increased by 14%, and the chip area is increased by 17%. The μAlpha is a very simple five-stage pipeline version of the Alpha architecture. For superscalar or out-of-order processors, the resource requirements will negligible compared to the total processor size.

Chapter 5

Related Work

5.1 Programmable Architectures

5.1.1 Microcode

Microcode is a hardware abstraction technique that was introduced mainly to simplify the control logic in a processor. It is used to implement ISAs on much simpler machines that can have different internal designs but still provide a unified architecture to programmers. Another benefit of using microcode is that fixing logical hardware errors can be done easily by reloading microcode.

The early CMOS S/390 processors used vertical microcode instead of the traditional horizontal microcode. The micro-instructions were very similar to the machine language instructions. They were kept on special storage and executed on separate processor chips to implement complex functions [16].

5.1.2 Millicode

Millicode is a type of vertical microcode with some differences. It was first introduced in S/390 G4 and then used in other generations including the IBM zSeries processors [16]. Millicode is very similar to normal code, however, some instructions and hardware facilities are only available to millicode. Unlike the prior models, It runs on a single chip and does not require distinct microprocessors. In addition to a special read-only cache, the millicode programs can be stored in the standard memory like a normal program. Millicode is used

to implement complex instructions, both for backward compatibility and for adding new functionality, like virtualization and transactional memory.

5.1.3 PALcode

A more programmable approach is offered by Alpha's Privileged Architecture Library code or PALcode. It was constructed to ease the transition from various preceding DEC architectures to a unified platform. That was done by conditionally extending the instruction set with custom calls depending on the OS being run [11]. Like millicode, PALcode helps to hide the internal details of the processor and support the operating system. Three versions of the PALcode were created, one for each OS, VMS, Digital UNIX, and Windows NT. PALcode is written in standard Alpha assembly with the addition of five new instructions that modify the privileged microarchitectural state.

PALcode allows developers to use standard programming tools with minimal modifications. It also executed very fast on Alpha processors. However, PALcode and Millicode are first and foremost tools for ISA architects. Their goal is compatibility across processor generations with complex operations that are still hardware-dependent since they were conceived as mechanisms for abstracting the microarchitecture. Millicode is not supposed to be programmed, similarly to microcode [16]. PALcode, on the other hand, is prescribed only for supporting DEC OSes and their slight variations [11].

5.2 Architectural Extensions

We discussed different Metal applications in Chapter 4. In the following sections, we overview some of the existing methods related to these applications. A more sophisticated and general version of these techniques can be built using Metal.

5.2.1 Protection Rings and IPC Mechanism

Multiple protection rings were first introduced by the Multics operating system which supported eight rings on the GE 645 and Honeywell 6180 [30]. Many modern computers have fewer rings. The X86 [20] and VAX [13] architectures have four rings of protection, and Alpha only has two [34]. The OpenVMS operating system on VAX uses all the four protection rings and the OS/2 on x86 uses three. However, most operating systems such as Windows NT [29] and Unix only use two privilege modes.

Porting an operating system to an architecture with a fewer number of rings is difficult because it requires emulating extra rings. However, the number of rings that the architecture supports can easily be modified with Metal so the problem of incompatibility between the OS and the architecture is solved.

Metal can implement different IPC mechanisms for transitions between address spaces or privilege levels. IPC mechanisms similar to Doors in Spring [15], Gates in HiStar [38], Call gates in Intel x86 [20], and Gates in Multics [7] can be built with Metal.

5.2.2 Capability-based Security

The IBM System/38 [18] and Intel iAPX 432 processors [26] are two examples of architectures that implement capability-based security in hardware using microcode. These systems were the results of a decade-long evolution of capability systems, with the 432 hardware being inspired by the Hydra operating system [6].

5.2.3 Secure Enclaves

In the last few years, there has been an increase in interest in secure enclave technologies. Most major architectures have created their version of this concept in hardware, each with slightly different specifications. Examples of these architectures include Intel SGX [8], AMD Secure Encrypted Virtualization [22], and ARM TrustZone [3].

Several research systems used software approaches to implement these mechanisms. Sanctum [9] and Sanctorum [25] built software enclaves with very little hardware support. These systems require a cryptographically secure random number generator, and optionally, memory encryption hardware to protect against physical attacks. A startup, PrivateCore, built a hypervisor that transparently encrypts memory to provide functionality like AMD SEV without hardware support [1].

5.2.4 Virtualization

Virtualization was first introduced in the IBM VM/370 [10] and implemented in numerous other architectures including the VAX [14], Alpha [23], POWER [19], Intel VT-x [21], AMD SVM [4], ARM [31], SPARC [36], and MIPS [28] processors.

The IBM zSeries virtualization extensions are mostly implemented in millicode [16], and Alpha hypervisor was built exclusively using PALcode [23]. Both of these systems

depend on their microarchitectures for virtualization. Using Metal, we can have our implementation of virtualization without any ties to hardware that works on any system.

Chapter 6

Conclusion

Metal is a novel programmable approach to privileged architecture. It provides a flexible privileged architecture that is programmable by software instead of being built in the hardware. It enables operating system developers to explore the design space and quickly prototype new privileged architectures.

Metal is inspired by PALcode and Millicode that both showed the practicality of this approach. However, unlike PALCode and Millicode that abstract microarchitectural details from the operating system and are used to implement custom privileged architectures in hardware, Metal is part of the architecture specification and enables operating system developers to implement privileged architectures using software.

The privileged architecture has evolved multiple times. To ensure backward compatibility, every time, a new set of primitives are added instead of modifying the existing ones. Metal adds a layer between microarchitectural layers and the operating system layer and can help to avoid such profusion. We believe that as Baumann [5] has noted, new complex architectural primitives should be expressed using software rather than hardware.

Protection modes such as security enclaves and virtualization extension are being added piecemeal to modern architectures. While this incremental approach is safer and isolates failures of a component from the others, this approach does not consider compatibility. Many protection mechanisms are composite entities that are built using finer-grained elements in microcode. Finding these common elements helps to achieve interoperability between these mechanisms. Furthermore, many security-oriented mechanisms have high complexity but since they are implemented in hardware, it is not straightforward to reason about the guarantees they provide. Using a framework such as Metal to implement these mechanisms helps to avoid such problems.

Metal also enables operating system developers to prototype new security models. Even though several privilege models have been proposed throughout the years, the Unix two-level approach (User-mode/Kernel-mode) has become the default because of simplicity and backwards compatibility across a wide variety of architectures. This two-level approach may not be the best solution in the cloud era. Metal provides a second chance for venerable ideas such as capability-based systems and partitioned operating systems (microkernels) to be evaluated for modern use cases.

In this thesis, we present the design of Metal, create a simplified version of Alpha architecture using Verilog, and implement Metal on this architecture. Implementing Metal is straightforward as it only requires a few basic blocks. We can add it to any microarchitecture with minimal hardware changes. The synthesis results show only a 14% increase in the number of cells after adding Metal to μAlpha. Furthermore, we show the practicality of Metal by discussing and implementing several different applications. We conclude that Metal works as expected in the use cases that we evaluated, and adding it to processors can significantly accelerate the development of operating systems and instruction set architectures.

References

[1] Privatecore Home Website. https://privatecore.com/. Accessed: 2019-06-07.

[2] Synopsys cell library. http://www.vlsitechnology.org/synopsys/vsclib013.lib.

[3] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security-enabling trusted computing in embedded systems (july 2004), 2014.

[4] Inc. AMD. Secure Virtual Machine Architecture Reference Manual. 2005.

[5] Andrew Baumann. Hardware Is the New Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

[6] Ellis Cohen and David Jefferson. Protection in the hydra operating system. *ACM SIGOPS Operating Systems Review*, 9(5):141–160, 1975.

[7] Fernando J Corbató, Jerome H Saltzer, and Chris T Clingen. Multics: the first seven years. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 571–583, 1971.

[8] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[9] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.

[10] Robert J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[11] Digital Equipment Corporation. *PALcode for Alpha Microprocessors: System Design Guide*. May 1996.

[12] Daniel W Dobberpuhl, Richard T Witek, Randy Allmon, Robert Anglin, David Bertucci, Sharon Britton, Linda Chao, Robert A Conrad, Daniel E Dever, Bruce Gieseke, et al. A 200-mhz 64-bit dual-issue cmos microprocessor. *Digital Technical Journal*, 4:35–35, 1993.

[13] Judith S Hall and Paul T Robinson. Virtualizing the vax architecture. In *Proceedings of the 18th annual international symposium on Computer architecture*, pages 380–389, 1991.

[14] Judith S Hall and Paul T Robinson. Virtualizing the VAX architecture. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 380–389. ACM, 1991.

[15] Graham Hamilton and Panos Kougiouris. *The Spring nucleus: A microkernel for objects*. Citeseer, 1993.

[16] Lisa Cranton Heller and Mark S Farrell. Millicode in an ibm zseries processor. *IBM Journal of Research and Development*, 48(3.4):425–434, 2004.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 289–300. ACM, 1993.

[18] Merle E Houdek, Frank G Soltis, and Roy L Hoffman. Ibm system/38 support for capability-based addressing. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 341–348. IEEE Computer Society Press, 1981.

[19] IBM Systems and Technology Group. *PowerISA Version 3.0 B*. March 2017.

[20] Inta Intel. Intel Architecture Software developer's Manual, Vol 1: Basic Architecture. *Available as ordering*, (243190):60056–7641, 2004.

[21] Intel Intel. IA-32 Architectures Software Developer's Manual Combined Volumes 2A and 2B: Instruction Set Reference. Technical report, AZ. Technical Report. Intel Corp, 2011.

[22] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[23] Paul A Karger. Performance and Security Lessons Learned From Virtualizing the Alpha Processor. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 392–401. ACM, 2007.

[24] Thomas F. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, August 4-6, 1986, Cambridge, Massachusetts, USA*, pages 105–112. ACM, 1986.

[25] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. Sanctorum: A lightweight security monitor for secure enclaves. *arXiv preprint arXiv:1812.10605*, 2018.

[26] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[27] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 175–188, 1993.

[28] MIPS Tech, LLC. *MIPS64 Architecture for Programmers Volume IV-I: Virtualization Module of the MIPS64 Architecture*. December 2013.

[29] M Pietreck. Windows internals: The implementation of the windows operation environment, 1993.

[30] Michael D Schroeder and Jerome H Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, 1972.

[31] David Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[32] Nir Shavit and Dan Touitou. Software transactional memory. In James H. Anderson, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213. ACM, 1995.

[33] Daniel P Siewiorek, Gordon Bell, and Allen C Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, Inc., 1982.

[34] Richard L Sites. Alpha axp architecture. *Communications of the ACM*, 36(2):33–44, 1993.

[35] Richard L Sites and Richard T Witek. *Alpha AXP architecture reference manual*. Digital Press, 2014.

[36] Sun Microsystems, Inc and Fujitsu Limited. *SPARC JPS2: Common Specification*. September 2003.

[37] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

[38] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. *Communications of the ACM*, 54(11):93–101, 2011.

APPENDICES

Appendix A

Supported Instruction Set

Conditional move	
CMOVEQ	Conditional move if reg = 0
CMOVNE	Conditional move if reg != 0
CMOVLT	Conditional move if reg <0
CMOVLE	Conditional move if reg <= 0
CMOVGT	Conditional move if reg >0
CMOVGE	Conditional move if reg >= 0
CMOVLBC	Conditional move if reg low bit clear
CMOVLBS	Conditional move if reg low bit set

Integer Computation	
ADDL	Add longword
S4ADDL	Add longword, scale by 4
S8ADDL	Add longword, scale by 8
ADDQ	Add quadword
S4ADDQ	Add quadword, scale by 4
S8ADDQ	Add quadword, scale by 8
CMPEQ	Compare signed quadword =
CMPLT	Compare signed quadword <
CMPLE	Compare signed quadword <=
CMPULT	Compare unsigned quadword <
CMPULE	Compare unsigned quadword <=
CMPBGE	Compare byte, unsigned
MULL	Multiply longword
MULQ	Multiply quadword
UMULH	Multiply quadword high, unsigned
SUBL	Subtract longword
S4SUBL	Subtract longword, scale by 4
S8SUBL	Subtract longword, scale by 8
SUBQ	Subtract quadword
S4SUBQ	Subtract quadword, scale by 4
S8SUBQ	Subtract quadword, scale by 8
AND	AND logical
BIS	OR logical
XOR	XOR logical
BIC	AND-NOT logical
ORNOT	OR-NOT logical
EQV	XOR-NOT logical
SLL	Shift left, logical
SRL	Shift right, logical
SRA	Shift right, arithmetic

Integer Branch	
BEQ	Branch if reg = 0
BNE	Branch if reg != 0
BLT	Branch if reg <0
BLE	Branch if reg <= 0
BGT	Branch if reg >0
BGE	Branch if reg >= 0
BLBC	Branch if low bit clear
BLBS	Branch if low bit set
BR	Branch
BSR	Branch to subroutine
JMP	Jump
JSR	Jump to subroutine
RET	Return from subroutine
JSR_COROUTINE	Jump to subroutine, return

Address/Constant	
LDA	Load address
LDAH	Load address high

Byte Manipulation	
EXTBL	Extract byte low
EXTWL	Extract word low
EXTLL	Extract longword low
EXTQL	Extract quadword low
EXTWH	Extract word high
EXTLH	Extract longword high
EXTQH	Extract quadword high
INSBL	Insert byte low
INSWL	Insert word low
INSLL	Insert longword low
INSQL	Insert quadword low
INSWH	Insert word high
INSLH	Insert longword high
INSQH	Insert quadword high
MSKBL	Mask byte low
MSKWL	Mask word low
MSKLL	Mask longword low
MSKQL	Mask quadword low
MSKWH	Mask word high
MSKLH	Mask longword high
MSKQH	Mask quadword high
ZAP	Clear selected bytes
ZAPNOT	Clear Unselected bytes

Load and Store	
LDL	Load sign-extended longword
LDQ	Load quadword
LDL_L	Load sign-extended longword, locked
LDQ_L	Load quadword, locked
STL_C	Store longword, conditional
STQ_C	Store quadword, conditional
STL	Store longword
STQ	Store quadword

Miscellaneous	
SEXTB	Sign extend byte
SEXTW	Sign extend word
CTPOP	Count population
PERR	Pixel error
CTLZ	Count leading zero
CTTZ	Count trailing zero
UNPKBW	Unpack bytes to longwords
UNPKBL	Unpack bytes to words
PKWB	Pack words to bytes
PKLB	Pack longwords to bytes